THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Lorna Patricia Shapiro

TITLE OF THESIS Sorting: A Survey, An Analysis

and Some Improvements

DEGREE FOR WHICH THESIS WAS PRESENTED M.Sc.

YEAR THIS DEGREE GRANTED 1973

Permission is hereby granted to THE UNIVERSITY OF

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Lorna Patricia Shapiro

TITLE OF THESIS Sorting: A Survey, An Analysis and Some Improvements

DEGREE FOR WHICH THESIS WAS PRESENTED M.Sc.

YEAR THIS DEGREE GRANTED 1973

THE UNIVERSITY OF ALBERTA

SORTING: A SURVEY, AN ANALYSIS
AND SOME IMPROVEMENTS

BY

LORNA PATRICIA SHAPIRO

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
AND RESEARCH IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE
EDMONTON, ALBERTA
SPRING, 1973

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled SORTING: A SURVEY, AN ANALYSIS AND SOME IMPROVEMENTS submitted by Lorna Shapiro in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

The purpose of this thesis is to examine current sorting techniques and consider how information can be used to increase their efficiency. A survey and classification of sorting algorithms is presented. Formal analysis criteria are applied to the algorithms, thereby obtaining relative efficiencies. The efficiency of algorithms as applied to partially sorted data or data containing redundant items is discussed. Also considered are means for utilizing known information about the data to increase sorting efficiency. In addition, it is shown that the theoretical lower bound on the number of comparisons required to sort a list containing redundancies is less than that for the same number of elements, but without redundancies.

ACKNOWLEDGEMENTS

I wish to thank Professor W. A. Davis, my supervisor, for his advice and guidance throughout the preparation of this thesis. The criticism offered by Professor L. W. Jackson and my fellow graduate students, Mr. B. J. Wesley and Mr. P. G. Rushton, is gratefully acknowledged.

The financial assistance received from the National Research Council of Canada, in the form of a scholarship, and from the Department of Computing Science, in the form of a teaching assistantship, is appreciated.

TABLE OF CONTENTS

Page

LIST OF TABLES

Chapter 1

Introduction

The purpose of sorting is to lend organization to a data base so that it is easy to extract the desired information, mechanically or visually. The great variety in data bases and in the organization required warrants the development of many approaches to sorting. Generally, these algorithms were designed under the assumption that there is no known information about the data base. However, for many sorting applications there is some such information available and it seems reasonable that, by making use of it, more efficient sorting can be achieved. The questions of interest, then, are what forms this information can take and how current techniques can be modified to gain more efficient sorting. To date, there has been very little published which concerns the problem of the use of information in sorting.

Before discussing individual sorting techniques, a number of terms which will be used throughout will be defined. Given a vector X of order N and a function K defined on each component, the problem of <u>sorting</u> is to determine the permuted vector Y such that [10]

$K(Y(i+1)) \geq K(Y(i))$, $(i=1,2,\ldots,N-1)$. The vector Y is said to be in ascending order on the key K. A descending order sort can be defined similarly.

The terms data items and items are used synonomously and indicate components of the vector to be sorted. The terms data item value and key value both refer to the value of the key function for a component of the input vector. A set of items is unique if their key values are all different and any two items are equivalent if their key values are the same. An item is a duplicate of another item if the two items are equivalent. N will be used to specify the number of items to be sorted. Whenever a log is used without specifying the base, it will mean base two. The symbol pairs, $\lfloor x \rfloor$ and $\lceil x \rceil$, are the greatest integer function and least integer function respectively;

i.e., $\lfloor x \rfloor$ satisfies $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$

and, $\lceil x \rceil$ satisfies $\lceil x \rceil - 1 < x \leq \lceil x \rceil$.

In order to provide background for the discussion of information use in sorting, the current sorting algorithms are classified and explained in Chapter 2. Analysis of these algorithms is discussed in Chapter 3 along with the results about the efficiency of specific algorithms. After examining the current 'state of the art', the problem of sorting efficiently by making use of known information is discussed in Chapter 4.

Chapter 2

A Survey

Sorting algorithms can be distinguished by the different types of file access required, and can be further classified by differences in methodology. Such a classification is presented, along with an explanation of specific algorithms.

Algorithms are classified as <u>serial</u> or <u>random-access</u> [2] depending upon whether or not serial-access files can be used throughout the process. Random-access algorithms are intended for use in the internal, rather than peripheral, memory of a computer and therefore are often referred to as internal sorts. Serial algorithms are generally performed on card, tape and disk files. Random-access techniques could be carried out with disk files, but, since random-access time for a disk is much greater than serial-access time, serial techniques are commonly used for such files [2].

2.1 Serial Sorts

There are four types of operations that can be carried out on serial-access files. They are differentiated

by the number of input and output files used and are defined [2] in Table 1.

Table 1. Serial Processes

Process	Input	Output
duplication	single	single
classification	single	multiple
merge	multiple	single
revision	multiple	multiple

Classification is the distribution of items from one file onto several files, while _simple_ _classification_ is a special case involving the allocation, to each output file, of a vector of consecutive items from the input file. Similarly, a _simple_ _merge_ is defined as a merge process after which each input file occurs as a vector of consecutive items in the output file.

With these definitions, two important subclasses of serial sorting methods--merge sorting and column sorting--can be described. _Merge_ _sorts_ consist of repeated applications of the pair of processes--simple classification and merge [10]. _Column_ _sorts_ are those algorithms consisting of repeated applications of the pair of processes--classification and simple merge [10]. In each case there is a pair of processes which, by repetition, produces a sorted output file. This pair of processes is called a _stage_ [10]

of the algorithm. A *phase* is the smallest process which looks at each item of the original input file [10]. The basic merge sort, for example, consists of two phases: simple classification and merge.

Partial pass algorithms are those which do not necessarily look at every item of the original input file at each stage [2]. The classification of serial sorting algorithms is shown in Figure 1. The methodology associated with each of these subclasses will now be considered and examples given.

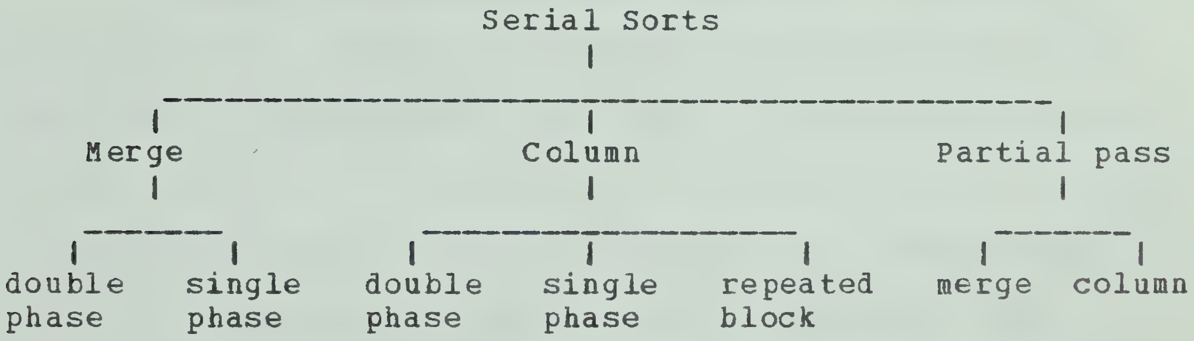


Figure 1. Serial Sorts

2.1.1 Merge Sorting

Given two files, each containing one string (*string* refers to a sorted list of items), a 2-way merge may be realized by first comparing the first item from each of the files and placing the smaller on the output file, thus

rendering this item 'unavailable'. Continuing in a like manner, the next two items available from the input files are compared and the smaller is placed on the output file. The merge is complete when no available items appear on the input files. If the input files each contain S strings then the process can be generalized to produce S output strings, each of which contains one string from both the input files. An m-way merge is a simple extension of a 2-way merge, using m input files. The process consists of repeatedly selecting the smallest of the m items next available (one from each file) for placement on the output file.

A <u>double phase</u> merge sort consists of repeated applications of simple classification and merge [10]. The input file is considered to be a set of strings, which may all be of length 1 if the file is in reverse order. A simple classification process places these strings onto m output files. If there are S strings, then no more than $\lceil S/m \rceil$ strings should be placed on any one output file in order to minimize the number of stages required. The m output files are then used as input to the merge phase. An example of a double phase 3-way merge is given on the following page. In this example, and others following, the input file is listed first.

16 72 | 19 30 85 | 49 75 | 13 | 12 63 | 21 45 | 17 90 | 35 46 77 |

simple classification :

16 72 | 19 30 85 | 49 75 |
13 | 12 63 | 21 45 |
17 90 | 35 46 77 |

merge :

13 16 17 72 90 | 12 19 30 35 46 63 77 85 | 21 45 49 75 |

simple classification :

13 16 17 72 90 |
12 19 30 35 46 63 77 85 |
21 45 49 75 |

merge :

12 13 16 17 19 21 30 35 45 46 49 63 72 75 77 85 90 |

In double phase merging, using strings occurring naturally within the input list results in variable length strings throughout the sort. There is a procedure called *string doubling* [10] whereby strings of length one are used for the initial classification. This results in strings of length $m, m^2, m^3, \ldots$ after the 1-st, 2-nd, 3-rd, ... merge phases. Because the lengths of strings are known, 'end-of-string' markers need not be used.

Another merge sort, the *single phase* merge, uses m input files and m output files at each stage. This sort varies from the double phase sort in that successive merged strings are placed on the successive output files in cyclic sequence. This classification process [10] is called

string classification and, along with the fact that each stage uses m inputs and m outputs, allows the classification and merge phases to be combined in one revision phase. An example of this process, using four files, is as follows:

16 72 | 19 85 | 49 75 | 65 | 12 63 | 50 | 17 90 | 35 77 |

string classification :

16 72 | 49 75 | 12 63 | 17 90 |
19 85 | 65 | 50 | 35 77 |

merge and string classification :

16 19 72 85 | 12 50 63 |
49 65 75 | 17 35 77 90 |

merge and string classification :

16 19 49 65 72 75 85 |
12 17 35 50 63 77 90 |

merge and string classification :

12 16 17 19 35 49 50 63 65 72 75 77 85 90 |

2.1.2 Column Sorting

Recall that column sorting is achieved by repetition of the pair of processes: classification and simple merge. Each classification phase uses one of the digits of the item key to distribute items onto different output files. Each digit is used once and successive digits are used in successive classifications--starting at the low order digit for the double phase column sort.

If the digital representation of the key is in base b, then b-way classification and b-way merge is used. Thus the process can be generalized to any vector key. An example of a double phase column sort on a 3 digit vector key in base 3 is as follows:

200,012,120,221,000,002,102,210,011

<u>classification</u> :

200,120,000,210
221,011
012,002,102

<u>simple merge</u> :

200,120,000,210,221,011,012,002,102

<u>classification</u> :

200,000,002,102
210,011,012
120,221

<u>simple merge</u> :

200,000,002,102,210,011,012,120,221

<u>classification</u> :

000,002,011,012
102,120
200,210,221

<u>simple merge</u> :

000,002,011,012,102,120,200,210,221

The single phase column sort is a modification of the double phase column sort [10]. At each stage, b input files and b output files are used. The explicit merge phase is eliminated by using, in turn, the b output files of the

previous classification as input to the present classification.

For both the double and single phase column sort, classification is carried out starting with the low order digit of the vector key. The repeated block column sort classifies the items first on the high order digit. One such classification produces b blocks of items which can be sorted separately and then combined in a simple merge to produce a sorted list. This method can be used to distribute the work load among a number of independent sorters [2].

The process could also be continued, working with successively lower order digits, to produce a sorted file [2]. There exists a serial sorting algorithm of this type which does not look at every item of the original input file at every stage and therefore will be discussed along with other partial pass algorithms.

2.1.3 Partial Pass Sorting

The two approaches to serial sorting just described have been modified to produce partial pass algorithms. The amphisbaenic sort and the partial pass column sort are both derivatives of column sorting techniques whereas the cascade, polyphase, and oscillating sorts result from modification of merge techniques.

Column sorting

The partial pass column sort is an algorithm that achieves the effect of one stage of a column sort for a base b key but uses fewer than b+1 files [10]. This algorithm, then, could be repeated q times for a vector key of length q to achieve a complete column sort. A series of partial passes is used, for each of which there is one input and one or more output files. After each partial pass, only the previous and next input files are rewound.

A zero-origin matrix, M, is used to describe the process [10]. M has one column for each partial pass and b+1 rows--one for each possible digit value and one to specify the input file at each pass. Entry M(i,j), i<b, indicates the output file to which keys with digit i, in the digit position being considered, are sent on the j-th partial pass. M is defined so as to minimize the pass fraction which is the total of the number of items looked at over all passes divided by the number of items.

An example of a partial pass column sort with 4 files for a decimal key is defined by M as shown in Table 2 [10]. The pass fraction in this example is:

$$2.8 = 1 + .4 + .5 + .5 + .3 + .1 .$$

Each of the six terms in this sum indicates, for one of the

partial passes, the fraction of the possible input values that are looked at.

Table 2. Partial Pass Column Sort Matrix

Digit	0	1	0				
	1	2		0			
	2	1	2	0			
	3	3			0		
	4	1	3		0		
	5	2		3	0		
	6	2		1		0	
	7	1	2	1		0	
	8	3			1	0	
	9	3			2		0
Input		0	1	2	3	1	2

The amphisbaenic sort is an example of a repeated block column sort using partial passes [10]. There are b+1 files required for a base b key. The algorithm uses a series of classifications (high to low order), and after each classification on a low order column, a simple merge of the last b sublists produced is performed. Classification from file i allocates:

digits 0,1,...,i-1 to files 0,1,...,i-1

digits i,i+1,...,b-1 to files i+1,i+2,...,b.

Files are written forward and read backward--except the initial input file, which is read forward. An example of this sort on a 2 digit, base 3 key is given on the next page.

```
Classification
Input:
   file  0   22,01,12,21,02,00,10,11,20
Output:
   file  1   01,02,00
         2   12,10,11
         3   22,21,20
Classification
Input:
   file  1   01,02,00
Output:
   file  0   00
         2   (12,10,11) 01
         3   (22,21,20) 02
Simple Merge
Input:
   file  2   (12,10,11) 01
         3   (22,21,20) 02
Output:
   file  0   00,01,02
Classification
Input:
   file  2   12,10,11
Output:
   file  0   (00,01,02) 10
         1   11
         3   (22,21,20) 12
Simple Merge
Input:
   file  1   11
         3   (22,21,20) 12
Output:
   file  0   (00,01,02) 10,11,12
Classification
Input:
   file  3   22,21,20
Output:
   file  0   (00,01,02)(10,11,12)  20
         1   21
         2   22
Simple Merge
Input:
   file  1   21
         2   22
Output:
   file  0   (00,01,02)(10,11,12)  20,21,22
```

Merge sorting

The cascade sort, a partial pass merge algorithm, uses m+1 files and initially distributes the strings onto m files of unequal length [15]. Each stage consists of a series of merges, starting with an m-way merge until one of the files becomes empty. Then an (m-1)-way merge is carried out on the files still containing strings, with the output going to the file just emptied. When another file becomes empty, an (m-2)-way merge is carried out. A 2-way merge completes the stage. Files are rewound after being emptied, and strings of variable length or of length one may be used initially.

If 4 files are used and the input tape consists of 31 strings, the distribution of strings on the files throughout the process is as follows [15]:

F1	F2	F3	F4
31	0	0	0
0	14	11	6
6	8	5	0
6	3	0	5
3	0	3	2
1	2	3	0
0	1	2	1
1	0	1	1
0	1	0	0

A slight modification of the cascade sort produces the polyphase sort [15]. This algorithm always uses merges of the highest possible order.

The string distribution throughout a polyphase sort with 31 initial strings and 4 files is as follows [11]:

F1	F2	F3	F4
31	0	0	0
0	13	11	7
7	6	4	0
3	2	0	4
1	0	2	2
0	1	1	1
1	0	0	0

The final partial pass merge sort to be considered is the oscillating sort [11]. This algorithm, while employing simple classification and merge phases, does not consist of repeated stages to produce a sorted file. Using m+2 tapes, m-way merges are carried out and it is assumed that the number of items in the initial file is a power of m. Unit length strings are used initially. In the first step of the process, unit strings are distributed onto m tapes and an m-way merge is carried out. This process is repeated until there are m strings of length m, at which point an m-way merge is used to produce a string of length m^2. Continuing in a like manner, m strings of length m^2 are produced and merged to form a string of length m^3. Files are written forward and read backward. The process is complete when all items are in one string. An example of this process with m=2 is found in Figure 2. Underscores are placed after strings resulting from a merge.

Step	F1	F2	F3	F4
1	3 5 2 7 1 4 8 6			
2	2 7 1 4 8 6		3	5
3	2 7 1 4 8 6	5 3		
4	1 4 8 6	5 3 2		7
5	1 4 8 6	5 3	7 2	
6	1 4 8 6			2 3 5 7
7	8 6	4		2 3 5 7 1
8	8 6		4 1	2 3 5 7
9			4 1 8	2 3 5 7 6
10		8 6	4 1	2 3 5 7
11	1 4 6 8			2 3 5 7
12		8 7 6 5 4 3 2 1		

Figure 2. Oscillating Sort

Note that the merged strings are in ascending or descending order according to whether their length is equal to an even or odd power of m. The merge process, therefore, must be able to merge in both directions.

2.2 Random-access Sorts

With large internal memory and virtual memory capabilities, in theory, few applications require external sorting, and even if external sorting is used, internal sorting of blocks of data is often used to produce long initial strings [16]. Algorithms designed for sorting data in internal memory will be dealt with in the following sections.

Internal sorting may be carried out on the actual file of items or on a surrogate of the file. A surrogate S of a file X(1 : N) is a data structure [16] that uniquely identifies each item of X. Sorting by surrogate is used when it is inconvenient to sort the items themselves. This may occur because the original items are long or of variable length, or the calculation of the key is difficult [16]. In the latter case, the key may be calculated once for each item and stored in the surrogate. The use of the actual file or a surrogate of the file is a difference in implementation rather than a distinction between algorithms. Random-access sorting algorithms are classified as shown in Figure 3.

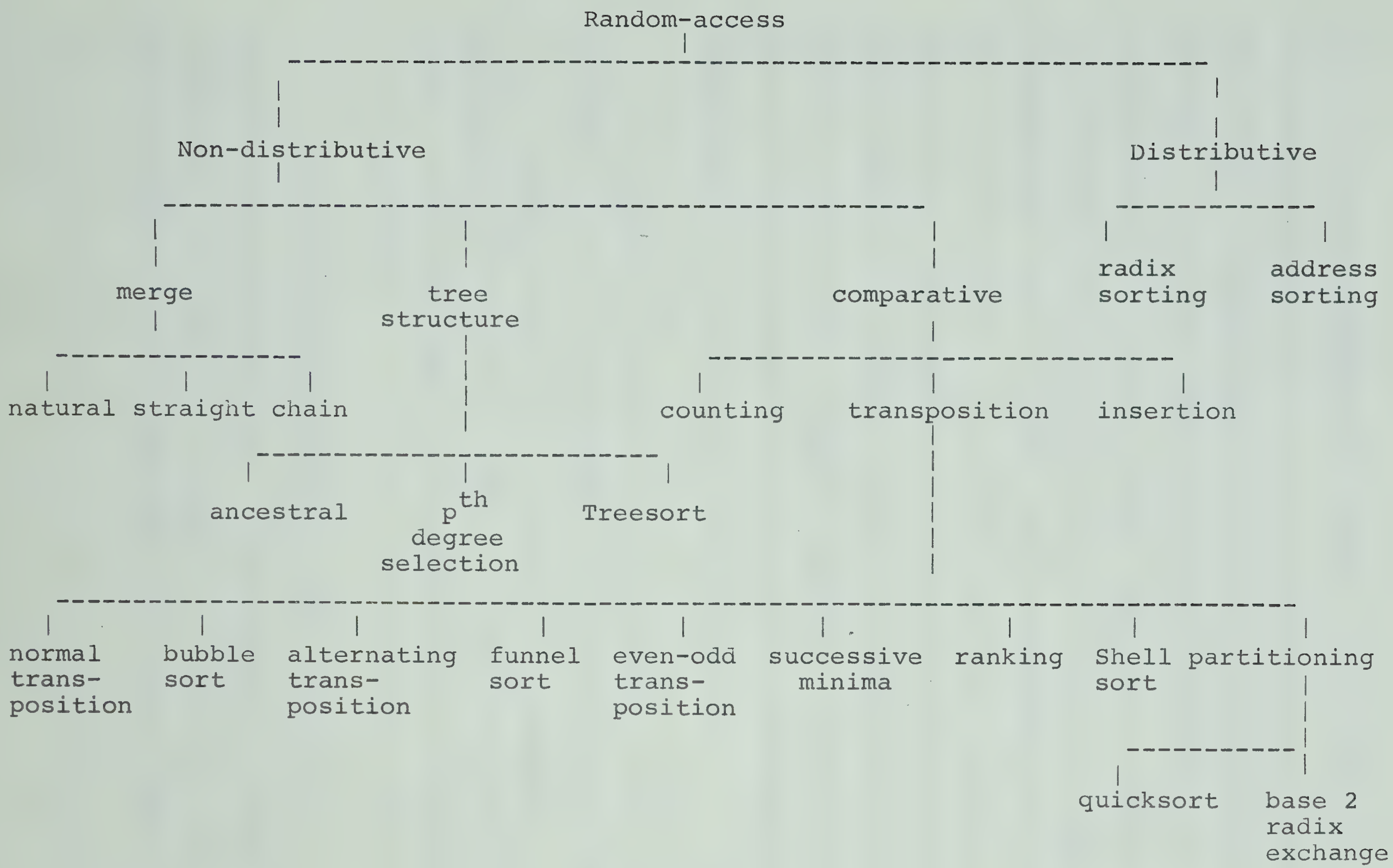


Figure 3. Random-access Sorts

In this classification scheme, algorithms are distinguished first on the basis of whether or not they are distributive. A *distributive* algorithm is one in which an estimate of the final position of each item is made by an examination of the item key itself. The item is placed in the estimated final position unless this location cannot accomodate the item, in which case an adjustment must be made. The adjustment is ordered or unordered depending upon whether or not the resultant list is necessarily in sorted order. However, the initial estimate is made solely on the value of the item key. A *non-distributive* algorithm is one in which the final position of each item is determined by comparison of that item with others in the list.

A possible characteristic of algorithms which may be important in certain applications is sequence preservation, or rank preservation [16]. An algorithm is *sequence preserving* if those items, whose key values are equal, appear in the same order in the sorted list as they occurred in the input list.

With this classification in mind, different methodologies associated with various classes of random-access sorts will now be discussed. The algorithms used for internal sorting will be explained and examples given.

2.2.1 Distributive Sorting

Inherent to the technique of sending an item to an estimated final location on the basis of its key value is the concept of considering available memory as a set of buckets, or bins, with items being assigned to one of these bins. There are several ways of implementing this concept [16].

Since it is possible that all N items could have the same key value, the case of all N items being sent to the same bin must be dealt with. One possible bin technique, then, is to allow for several blocks of memory; each block representing one bin and having a size equal to the length of N items [15]. This technique is extremely expensive in terms of storage space. A similar bin technique is the use of M blocks of core, each having the size to accomodate Q items, such that $M \times Q \geq N$ [16]. In this case, an adjustment must be made if more than Q items are sent to the same bin.

Linear arrays, with each item position representing a bin, may be used to implement the bin technique [15]. With bins of size one, an adjustment algorithm is again needed to handle the case in which more than one item is sent to any bin. Such an algorithm may simply involve placing the item in the nearest vacant bin, or may involve a scan (linear or binary) for the proper position of the item and a shift of

items to accommodate the new item [16]. Adjustment may involve only shifts in one direction, or shifting in either direction may be allowed for, choosing the one which involves fewer items. When a linear array is used, a decision as to the size of the array must be made. The larger the array size, the smaller is the possibility of conflict.

The final bin technique to be discussed is the use of linked lists, one for each bin [15]. The first item in each linked list is simply a pointer to a list. As each item is sent to a bin, it is inserted at the end of the linked list associated with that bin. A final sorting of items within each linked list may be necessary.

The choice of a bin technique is dependent upon such factors as the storage space available and the distribution of values expected. The following descriptions of address and radix sorting are given without further consideration of the choice of bin technique.

Address sorting

Address (or address calculation) sorting involves the use of a function which maps key values into bin locations [16]. For each input item, the function is calculated and the item is sent to the corresponding bin. An adjustment algorithm may be necessary as well as a final

collection process to produce a sorted list. The effectiveness of address sorting is largely determined by the suitability of the mapping used.

Radix sorting

Radix sorting is analagous to column sorting [16]. For each item, the key is considered as a q digit vector in base b. There are q distributions of the items carried out on the basis of successive digits of the key and b bins are used.

Upward radix sorting starts with the lowest order digit, using successively higher order digits on successive distributions [16]. Effectively, a simple merge is carried out on the bins between each classification. An example of upward radix sorting on 2 digit keys in base 3 is as follows:

21,22,02,12,11,01,22,02,11,20,12

Stage

1 (20)(21,11,01,11)(22,02,12,22,02,12)

2 (01,02,02)(11,11,12,12)(20,21,22,22)

Two distributions produce a sorted list.

Downward radix sorting distributes on the q digits successively from highest to lowest order [16]. Each bin of items produced on the i-th distribution is dealt with

separately on the (i+1)st distribution $(0<i<q)$. An example of such a sort on 3 digit keys in base 3 is as follows:

212,220,102,121,112,022,202,021,111,220,122

Stage

1 (022,021) (102,121,112,111,122) (212,220,202,220)
2 (022,021) (102) (112,111) (121,122) (202) (212) (220,220)
3 (021) (022) (102) (111) (112) (121) (122) (202) (212) (220,220)

For both address and radix sorting, an estimate of the final position of each item is made by examination of the item value. Item comparisons may be necessary for address sorts but are not used for radix sorting.

2.2.2 Non-distributive Sorting

Recall that non-distributive algorithms are those which derive the final sorted position of an item by comparison of items. Non-distributive algorithms are classified as *merge* sorts, *tree structure* sorts or *comparative* sorts depending upon whether merge techniques, tree structure techniques, or neither of these are used.

Merge sorting

Internal merge sorting is essentially the same process as serial merging and is generally restricted to 2-way merging [2]. The method of choosing the initial strings differentiates the three types of internal merge

algorithms--natural, straight and chain merging [16].

The natural merging algorithm uses as initial strings the set of maximal ordered sublists in the input list [16]. If there are p such strings, then $\lceil \log p \rceil$ stages are required to produce an ordered list. At each stage, pairs of adjacent strings are merged. Because the initial and subsequent strings are of variable length, pointers or end-of-string markers are required.

The straight merging algorithm uses initial strings of length one [16]. The merging process is as described for natural merging. This procedure has the advantage of largely obviating the necessity for pointers or end-of-string markers since all strings, with the possible exception of the last, are of the same length. In place of these markers, counting can be used.

The chain merging algorithm uses, as initial input strings, maximal ordered subsequences (called chains) occurring in the input list [16]. The relative positioning of items in a subsequence is the same as in the initial list, but adjacent items of the subsequence need not be adjacent in the input list. For example, the input list 2,1,4,3,6,5,8,7 contains two maximal subsequences, 2,4,6,8 and 1,3,5,7. The chains from an input list are found as follows. The first item starts the first chain. If the

second item is larger than or equal to the first, it is added to the first chain: if not, then it is used to start a second chain. Each new item is placed on the first chain possible, but if it is less than the last item of every chain then a new chain is started. Since the sequence of largest items of each of the chains is monotone increasing, items can be placed using a binary scan. Having derived the initial input strings, a series of 2-way merges is carried out. The two consecutive strings of smallest total length are always merged. The algorithm is complete when only one string remains.

Tree structure sorting

Throughout the discussion of tree structure and comparative sorting algorithms, the term *iteration* will be used to indicate a process that, upon repetition, produces a sorted list. Sorting algorithms utilize tree structures for insertion, selection, and transposition techniques. *Insertion* is a technique whereby items of the input list are inserted, one at a time, in sorted position in a partial sorted list. A *selection* sorting process involves selecting from the list, at the i-th iteration, the smallest of the remaining N-i+1 items. *Transposition* is a technique whereby pairs of items are exchanged in the list to produce a sorted list.

The *ancestral* sorting algorithm uses a binary tree structure for an insertion technique [16]. Each non-terminal node contains an item as well as a left and/or right pointer to a node whose item key is correspondingly less than or greater than the item key of the node concerned. The first item received forms the item value of the first node. For each new item, a search is made of the tree for its proper position by comparison of the item key with keys of node items and following the tree along the left or right pointer depending upon the result of the comparison. When a terminal node, T, is reached, the new item forms a new node and a pointer (left or right) to this new node is placed in T.

The set of tree structure selection algorithms are referred to as *p-th degree selection* sorts [2]. The general description of the tree for a p-th degree selection sort is as follows. The tree consists of p+1 levels (0 to p). Level 0 is the root node and level p consists of N leaf nodes; one for each item in the list to be sorted. Ideally, the tree has a branching factor of m, where $m^p = N$. The tree is constructed, from level p-1 to level 0, by associating with each node the minimal valued item of its m successor nodes. Thus, the least item value is associated with the root node.

At each iteration of the algorithm, the item value, X, associated with the root node, is placed in an output list. At the p-th level, X is replaced by an infinite key so

as to remove the item from further consideration. X must also be replaced at levels p-1,p-2,...,0. In each case, a new minimum is selected from the m filial nodes previously represented by X. In N iterations the algorithm produces the output of a sorted list.

Specific cases of the general p-th degree selection sort are the quadratic sort, the cubic sort, and the tournament sort. The <u>quadratic</u> sort [2], a second degree selection sort, uses a tree with a branching factor of b, where b^2= N. The <u>cubic</u> sort is a third degree selection sort [2] and uses a tree with a branching factor of b, where b^3 = N. The <u>tournament</u> sort [2] uses a binary tree and thus is a p-th degree selection sort, where $p = \lceil \log N \rceil$ [2]. An example of the first three iterations of a tournament sort, with N = 8, will serve to demonstrate the p-th degree selection technique. The letter z is used to indicate an infinite key value.

Iteration 1

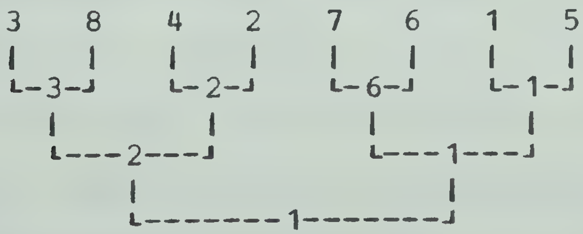
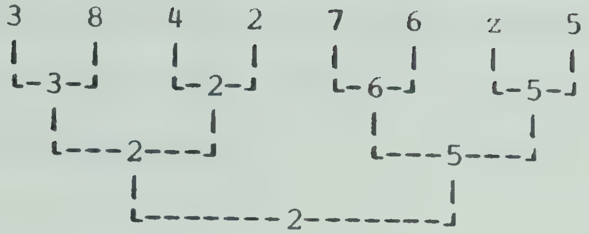


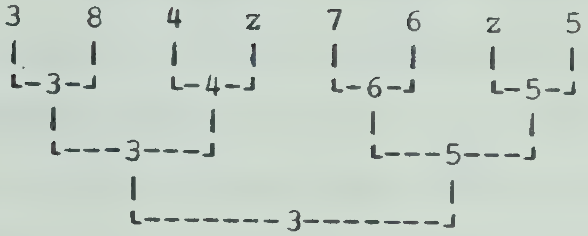
Figure 4. Tournament Sort

.../continued on page 28

Iteration 2



Iteration 3



.
.
.

Figure 4. Tournament Sort (continued)

The treesort algorithm [16] is a transposition sort using a tree structure. Rather than building a tree, the linear input array X(1 : N) is considered as a binary tree. The root node of the tree is X(1), and for i ≤ N/2, X(i) is considered as the predecessor of X(2i) and X(2i+1). Initially, exchanges are made along the paths of the tree so defined until every non-terminal node, P, except the root node, roots a subtree containing no item with key value greater than the key value of P. Having established this descending tree, a series of N-1 iterations is made. At the i-th iteration, the largest of the N-i+1 remaining items is

brought to the root node. The descending property is maintained. The root item is then exchanged with the (N-i+1)th item of X(1 : N). After N-1 iterations the list is in sorted order.

Comparative sorting

Throughout the discussion of comparative sorts, the term pass is used to indicate a sequence of comparisons involving some or all of the data items. An iteration may involve one or more pass. The first class of comparative sort algorithms to be discussed is the class of counting sorts, i.e., those which use a count vector of length N [16]. This vector is initialized to 1 and a series of N-1 sets of comparisons is made. In the i-th set of comparisons ($1 < i \leq N$), item i is compared in turn with items 1,2,...,i-1. For each comparison, the count vector item corresponding to the larger item is increased by 1. After all comparisons are made, the count vector indicates the sorted order positioning of items. The items may then be ordered in place by transposition, or may be copied to an output file in proper sequence.

Insertion sorting is another comparative algorithm [15]. The first item in the list is inserted into a new list and considered as a sorted partial list of length one. When

the (i+1)st item is to be inserted, a search of the sorted partial list of size i determines its proper position. Items of the partial list are then shifted to accomodate the (i+1)st item. The search technique may be linear or binary and the size of the list used for insertion may be N or larger.

The final class of comparative non-distributive sorting algorithms consists of the set of transposition sorts [16], i.e., those which order the list by a sequence of comparisons and exchanges. These algorithms all employ several passes over all, or some of, the data. The term 'comparison and exchange' is used to mean that if items in positions i and i+j (for j>0) are compared and the key value of the latter is less than that of the former, an exchange is made.

The normal transposition sort consists of several iterations, each consisting of one pass over the entire list [16]. During each pass, a series of comparisons and exchanges are made between items in positions (1,2) (2,3)...(N-1,N), in that order. When an iteration yields no exchanges, the list is in sorted order. A slight variation of this is the bubble sort which takes advantage of the fact that, after i iterations, the largest i items are in place and need not be considered [16]. Thus, for the i-th iteration, a pass consists of the series of comparisons

between item pairs (1,2)(2,3)...(N-i,N-i+1). Again, an iteration with no exchanges halts the process.

For the alternating transposition sort [16], each iteration involves two passes, each over the entire list but in alternating directions. The first pass is the same as normal transposition; the second pass is defined by the sequence of comparisons (N-1,N),(N-2,N-1) ...(1,2). The funnel sort [16], a variation of the alternating transposition sort, uses the fact that, after i passes, the $\lceil i/2 \rceil$ largest items and the $\lfloor i/2 \rfloor$ smallest items are in sorted position. These items are therefore not considered in further passes.

The even-odd transposition sort [16] again uses iterations consisting of two passes over the data. The two passes are defined, for N even, by the series of comparisons:

(1,2) (3,4) ... (N-1,N) and
(2,3) (4,5) ... (N-2,N-1).

A pass with no exchanges halts the process.

The successive minima algorithm [16] is a transposition sort requiring N-1 iterations. On the i-th iteration, the minimum of items in positions i, i+1,...,N is found and exchanged with the item in position i.

The ranking sort [16] is an insertion sort carried out by transposition. This algorithm consists of N-1 iterations. At the i-th iteration, the first i items are considered as a sorted partial list and the (i+1)st item is inserted into sorted position within this list. The insertion is achieved by a sequence of comparisons and exchanges of item pairs (i,i+1) (i-1,i) ... (1,2) in that order. When a comparison in this sequence yields no necessary exchange, the (i+1)st item has been properly inserted and the next iteration is begun.

The Shell sort consists of a series of iterations, where, during the i-th iteration, a ranking sort is carried out on d(i) lists, headed by items in positions 1,2,...,d(i) and consisting of items a distance d(i) apart [16]. The common choice for the set of d(i), Hibbard's modification, is defined as follows:

if $2^k < N \leq 2^{k+1}$

then $d(1) = 2^k - 1$ and $d(i+1) = (d(i)-1)/2$.

This choice of d(i) generally results in fewer comparisons than Shell's initial choice of d(i) which was:

$d(1) = N/2$

$d(i+1) = d(i)/2$.

Another scheme [16] for choosing d(i) has also been suggested. The first distance used is the nearest odd integer greater than N/4, if N is greater than 15, and N/2

otherwise. The choice of d(i+1) is determined in the same manner, using d(i) instead of N.

The final comparative technique to be considered is partitioning which is the process of dividing each list into sublists [15] during every iteration. Two such algorithms are the base 2 radix exchange sort and quicksort, of which there are several modifications.

The base 2 radix exchange sort [15] requires q iterations to sort a list of q digit binary numbers. The process is analagous to a downward radix sort but, since each bin is distributed into only 2 bins, an exchange technique is employed for the partitioning process. At the i-th iteration, each non-trivial, i.e., non-unit length, sublist is partitioned into a sublist of items with 0 in the i-th digit followed by a sublist of items with 1 in the i-th digit. Partitioning is carried out with two pointers, called front and back, which are initially positioned at the beginning and end of a sublist and move towards each other. When the front pointer finds an item with 1 in the i-th digit, the back pointer looks for an item with a 0 in that digit. The two items are then exchanged. The partition is complete when the pointers cross. After q iterations the list is sorted.

The quicksort algorithm [16] and modifications thereof, also partition each non-trivial sublist using

pointers as does the radix-exchange sort. A median estimate, M, is chosen randomly from each sublist. A scan and exchange process divides each sublist into 3 sublists containing only items $\leq M$, $=M$, $\geq M$ respectively. The middle sublist contains items in sorted position and is not dealt with further. The process is iterative and the shorter of the two sublists is always dealt with first to save on storage space required for segment delimiters. A sublist of size two is ordered by comparison and exchange.

The variations of quicksort stem from the different methods of choosing a median estimate. _Quickersort_ uses the middle item of a sublist as the median estimate [16]. _Samplesort_ uses as a median estimate the median of a sample of items drawn from the sublist [16]. The sample may be chosen randomly or may consist of items at regularly spaced intervals. A sample may be chosen from each sublist or one sample from the initial list may be taken and used to provide median estimates. If a bound on the key values is known, say 2^k, the _general radix exchange_ sort [15] may be used. This is a modification of quicksort using the median, the quartiles, etc., of the range 0 to 2^k as median estimates. Van Emden's modification [16] starts with an upper and lower bound on the median estimate and alters these as necessary to ensure that no item in the segment has a key value between them.

Sorting algorithms are distinguished first on the basis of whether they are intended for implementation using serial or random-access files. Serial algorithms are further classified as merge, column or partial pass by differences in methodology. The classification of random-access algorithms as distributive or non-distributive is based on whether or not the item value itself is used to obtain an estimate of the final location. The merge, tree structure and strictly comparative approaches to non-distributive sorting have been explained. Throughout this discussion, specific algorithms were examined and related to this classification.

Chapter 3

An Analysis

In this chapter sorting algorithms are analyzed and compared. Because of the wide variety of applications of sorting, it is impossible to state that one algorithm is best. For example, the oscillating sort and polyphase sort may not be directly compared if the computer, on which they are to be run, does not have tape devices that read backward. Nevertheless, the commonly accepted measures of performance for the different classes of algorithms are considered and known results are tabulated in this chapter. When possible, the best within classes of algorithms are selected.

3.1 Measures of Analysis

Analysis of serial sorting algorithms is based on two measures [10]:

1. the number of files required, and
2. the execution time.

In comparing efficiency the number of files is to be considered fixed. It is commonly assumed that the execution time of a serial process is spent mainly in reading and

writing of files and therefore time is directly proportional to the required number of passes of the file [10]. The relationship between the number of passes and the actual execution time is determined [10] by factors such as the average length of the items, the reading and writing rate of serial files, the rewinding time and, possibly, overhead in going from a forward to a backward operation or vice versa. The number of passes can be used as a measure of efficiency for this type of algorithm.

On the other hand, for random-access algorithms there are three measures of analysis used [10]:

1. the number of comparisons necessary,
2. the number of data moves or transpositions needed,
3. the storage ratio, C/N, where:

   C denotes the storage space, measured in number of items, allowed for data or pointer space to sort N items.

Sometimes a measure called the <u>scan length</u> [10], which is the total number of items examined throughout the sort process, is used instead of the number of comparisons. Scan length is of value when analyzing distributive algorithms but, for non-distributive algorithms, a constant of proportionality, depending on the algorithm but not on the number of items relates the scan length and the number of comparisons. Thus, only the latter need be considered.

A theoretical lower bound of N log N has been established [12] for the maximum number of comparisons required to sort N non-redundant data items. The 'possible outcomes' argument, used to establish this lower bound, states that, for N numbers, there are N! different permutations that may result from the application of any sorting algorithm. These may be considered as the leaves of a computation tree, which is a theoretical model used to represent the procedure for finding the solution for some problem [12]. Each non-terminal node of the tree represents a computation and has as many branches leaving it as there are results to the computation. Each terminal node represents a possible solution and, thus, each root-to-leaf path represents a sequence of computations for arriving at a solution. The minimum depth of such a tree is the theoretical lower bound on the number of computations required to always produce a solution. Generally, the computation tree for the sorting problem is binary, each node representing a comparison of two items. Since such a tree must have N! leaves, it must be of depth log N!. Using Stirling's approximation, $\log N! \doteq N \log N$. This, then, is the theoretical lower bound on the number of comparisons required to sort any N non-redundant items.

In addition to the quantitative measures used in analyzing algorithms the following qualitative factors must be considered:

1. Is the algorithm sequence preserving?
2. Does the algorithm take advantage of any sorted order present in the initial list?
3. Does the algorithm require fewer comparisons and data moves when duplicate values are present in the list?
4. Can the algorithm accept parameters regarding initial ordering or duplications and use the information to sort more efficiently?

3.2 Analysis of Serial Algorithms

In analyzing serial algorithms there are some considerations with respect to the efficient use of tapes that should be discussed. An in depth analysis of these factors is available [11] and only those of major importance will be mentioned here.

Tape device characteristics influencing sorting time are the data transfer rate and the rewind speed. The backward read capability is assumed for some algorithms and, if available, can be incorporated into other algorithms to increase sorting speed.

When implememting serial sorts using tapes, a large block size is desirable in that less waste space results. However, internal memory restrictions on the buffer size is a limiting factor. The use of tape devices on different channels and the overlapping of tape rewinds with other processes both result in greater sorting efficiency.

Merge Sorting Algorithms

Neither normal nor partial pass merge sorts are sequence preserving. Merge techniques make use of inherent sorted order in the initial list if natural strings are used but not if string doubling is used. The expected number of maximal strings (a maximal string is a string not contained in any larger string) within a list is [2]:

$$(N + 1 - E(D))/2$$

where E(D) is the expected number of duplications. Thus, duplicate items increase the expected string length.

Letting S signify the number of initial strings and considering the case in which there are 2m files, where $m>1$, the double and single phase merge sorts require respectively $2 \lceil \log_{2m-1} S \rceil$ and $\lceil \log_m S \rceil$ passes [10]. It is because the number of passes is dependent upon S that merge sorts can be considered to take advantage of inherent order and of duplicate items. Note that the ratio :

$$(2 \lceil \log_{2m-1} S \rceil) / \lceil \log_m S \rceil$$

assymptotically approaches 2 as S becomes large, indicating that single phase merge sorting is more efficient than double phase [10].

Column Sorting Algorithms

Column sorting algorithms, normal or partial pass, require the same number of passes whether or not duplicate items are present. No use is made of inherent order within the initial list, nor can such information, if made available, be used. In general, column sorts are sequence preserving.

Letting g signify the largest key value, if there are 2b files then the double and single phase column sorts require $2\lceil\log_{2b-1} g\rceil$ and $\lceil\log_b g\rceil$ passes respectively. As in the case of merge sorting, the ratio of these approaches 2 [10], making single phase column sorting preferred to double phase. The effect that the base chosen has on the number of passes is given by the following. A vector key of maximum value g can be considered as q_1 digits in base b_1 or q_2 digits in base b_2. A single phase column sort would then require either q_1 or q_2 passes and $2b_1$ or $2b_2$ files. The relationship between the number of passes is as follows [10]:

$$\frac{q_1}{q_2} = \frac{\log_{b_1} g}{\log_{b_2} g} = \frac{\log_g b_2}{\log_g b_1}$$

Partial Pass Sorting Algorithms

The partial pass column sort described in Chapter 2 used 4 files to sort a base 10 vector key. If g is the range of the key, then there are $\lceil \log_{10} g \rceil$ digits and, since the pass fraction for the method described was 2.8, the total number of passes required is $2.8 \lceil \log_{10} g \rceil$. This compares favorably with both the base three double phase column sort and the base two single phase column sort which could also be performed with 4 files [10].

The amphisbaenic sort is sequence preserving if q, the number of digits, is odd, but for q even, reverses the initial sequence of equal valued items. Each item is classified q times and is merged at most once. Therefore less than q+1 passes are used in total [10]. There are b+1 files required for a base b key.

The cascade sort requires that initial strings be distributed onto tapes using a 'perfect distribution'. An algorithm for calculating perfect distributions is available [11]. Table 3 gives a set of perfect distributions for six tapes.

Table 3. Perfect Distributions for Cascade Sort

Number of Strings	F1	F2	F3	F4	F5	Number of Passes
1	1	0	0	0	0	0
5	1	1	1	1	1	1
15	5	4	3	2	1	2
55	15	14	12	9	5	3
190	55	50	41	29	15	4
671	190	175	146	105	55	5
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
	a	b	c	d	e	n
	a+b+c+d+e	a+b+c+d	a+b+c	a+b	a	n+1

If the number of strings, S, does not equal the sum of a perfect distribution, then dummy strings are used to increase S to the next acceptable total. For 671 strings, the double phase merge would require $2 \lceil \log_5 671 \rceil = 10$ passes rather than 5 and the single phase merge would require $\lceil \log_3 671 \rceil = 6$ passes.

The polyphase sort also requires a 'perfect distribution' of initial strings, although it is somewhat different than that of the cascade sort. The distribution method is illustrated in Table 4 for the case where there are 4 files [11].

Table 4. Perfect Distributions for Polyphase Sort

Number of Strings	F1	F2	F3	F4	Number of Passes
1	0	0	1	0	0
3	1	1	0	1	1
5	2	0	1	2	2
9	0	2	3	4	3
17	4	6	7	0	4
31	11	13	0	7	5
57	24	0	13	20	6
105	0	24	37	44	7
193	44	68	81	0	8

The polyphase differs from the cascade sort in that initial strings are processed an unequal number of times. Some results with regard to the average number of times strings are processed, i.e., the number of passes, are shown in Table 5 [11].

Table 5. Pass Fraction for Polyphase Sort

Number of Strings	Number of Tapes	Passes
100	3	7.2
100	6	3.3
1000	3	10.8
1000	6	7.2
5000	3	13
5000	6	6.6

The double phase merge requires $2 \lceil \log_5 1000 \rceil = 10$ passes to sort 1000 strings using 6 tapes as opposed to approximately 7.2 passes for the polyphase sort. The single phase merge requires $\lceil \log_3 1000 \rceil = 7$ passes.

In analyzing the oscillating sort it is apparent that any item in a merged string of length m, m^2, m^3 has been processed 2, 3, 4 times respectively. Thus, if $x = \lceil \log N \rceil$, then x+1 passes are required for the oscillating sort using m+2 files.

The analysis of serial sorting algorithms indicates that, for column sorting, single phase is more efficient than double phase for any fixed number of files. The partial pass column sort is less efficient than single phase sorting in that more passes are required, but it may be used to save on files. The amphisbaenic sort uses fewer files, and approximately the same number of passes as single phase column sorting and thus is the most efficient. However, both forward and backward read capabilities are necessary to implement this algorithm.

Single phase merge sorting is more efficient than double phase. The partial pass merge algorithms are generally more efficient than single phase merge sorting. Of these, the oscillating sort is the most efficient [11] but it requires forward and backward reading. The cascade and polyphase algorithms do not require backward read. The polyphase sort is the more efficient for fewer than seven files, but for more than six files the cascade sort is more efficient [11].

The choice between column and merge sorting is dependent upon the form of the key value and the expected distribution and ordering of these values. It is the base chosen to represent key values that determines the number of files, and to a certain extent, the efficiency of a column sort. If base conversion is necessary, but complex, in order to implement a column sort then merge sorting should probably be used. As the likelihood of redundancies and sorted order within the input list increases, the desirability of merge sorting becomes pronounced. Merge algorithms also offer a greater choice in the number of files to be used and this may be advantageous if there is much variance in the number of files that may be availbble at any time.

3.3 Analysis of Random-access Algorithms

As was stated in Chapter 2, virtual memory capabilities have tended to reduce the necessity for serial sorting. Due to the common usage of internal sorting algorithms, the analysis and comparison of their efficiency is well studied.

Distributive

For distributive sorting algorithms, scan length is a more meaningful measure than the number of comparisons

because, while the scan length must be at least N, the number of comparisons can be 0. The efficiency of address sorting is dependent on both the address calculation function and the storage ratio. As the storage ratio increases, the expected number of data moves decreases but the compaction time increases. Experimental results [16] have indicated that optimum storage ratios are approximately 2.4 and 1.8 for ordered and unordered adjustment algorithms respectively. Address sorting is sequence preserving.

The other distributive algorithms, upward and downward radix sorts, are sequence preserving and not sequence preserving respectively. In both cases, the storage ratio is approximately 2. For upward and downward radix sorting, the scan length and the number of transfers are both Nq (where q is the number of digits).

Initial ordering present within the input list does not increase the efficiency of distributive algorithms. For address calculation sorts, a non-uniform distribution of redundancies can result in a great increase in the number of comparisons. However, information regarding the distribution of redundancies can be used to determine a good address function and bin allocation scheme.

Merge algorithms

Internal algorithms are sequence preserving and the natural and chain sorts both operate more efficiently when there is some ordering of the input list because this reduces the number of initial strings. Merge sorting algorithms usually use 2-way merging and a storage ratio of 2 [2]. Analysis results for the three merge techniques, assuming unique items [16], are found in Table 6.

Table 6. Analysis of Merge Sorts

	Scan Length			Item Transfers		
	Expected	Max	Min	Expected	Max	Min
Natural	Nlog(N/2)	NlogN	N	Nlog(N/2)	NlogN	N
Straight	NlogN	NlogN	NlogN	NlogN	NlogN	NlogN
Chain	*NlogN	NlogN	N	*NlogN	NlogN	N

* - experimental result

Duplicate items increase the expected string size for natural and chain merging and, thus, lead to an increase in sorting efficiency. Merge algorithms cannot readily utilize information regarding initial ordering or the distribution of duplicate items.

Tree Structure Algorithms

The tree structure sorting algorithms do not take advantage of initial ordering. Ancestral sorting is least efficient for a sorted input list. Redundancies do not

increase the efficiency of the ancestral and p-th degree selection sorts which are sequence preserving while treesort is not. Duplications tend to make the treesort more efficient in that the descending characteristic is more easily achieved. None of these algorithms can utilize information regarding initial ordering or the distribution of duplications. Analysis results for these algorithms are found in Table 7 [16].

Table 7. Analysis of Tree Structure Sorts

	Comparisons			Data Moves			S.R.
	Expected	Max	Min	Ex	Max	Min	
Ancestral	O(NlogN)	N(N-1)/2	O(NlogN)	N	N	N	4
p-th Degree Selection	(m-1)pN	(m-1)pN	(m-1)pN	pN	pN	pN	(Nm-1)/ N(m-1)
Treesort	*O(NlogN)			*O(NlogN)			1

* - experimental result (further results not available)

Counting

Counting sort algorithms are sequence preserving. The storage ratio is 2 or 3, depending upon whether 1 or 2 copies of the file are used. Generally, two copies are used and, thus, N data moves are required [16]. If one copy is used, from 0 to N exchanges are required. Note that an exchange generally requires three data moves. In either

case, N(N-1)/2 comparisons are used to determine the count vector. Neither redundant values nor sorted order simplify the counting process. If it is expected that few items will be out of sorted position then the use of one file with ordering by exchanging is preferred.

Insertion

Ranking by insertion is a sorting technique that takes advantage of initial ordering but does not necessarily perform better when redundancies are present. Parameters regarding order or distribution of duplications cannot be used to increase the efficiency. Generally, the list used for insertion is of length N or 2N and thus the storage ratio is 2 or 3 [15]. Ranking by insertion is sequence preserving. If an insertion vector of length N is used, then the average number of data moves is $O(N^2)$. For a vector of length 2N, about one half as many data moves are required. A linear scan and a binary scan result respectively in $O(N^2)$ and $O(N\log N)$ comparisons.

Transposition

All of the transposition sorts except the Shell and partitioning algorithms are sequence preserving. Generally, inherent sorted order in the input list results in fewer transpositions and, for those algorithms which do not have a

predetermined number of passes, can result in the use of fewer than the maximum number of passes. The efficiency of transposition sorts can be expected to increase slightly when duplicate values are present--largely because fewer transpositions are necessary. For example, the even-odd transposition sort requires, on the average, 5.5 comparisons and 3 transpositions for a list of 4 non-redundant items. The same algorithm, operating on a list of 4 items of which 2 are unique and appear twice, requires an average of 5.1 comparisons and 2 transpositions. Knowledge of the distribution of data values can be used to provide good median estimates for the quicksort algorithm.

The storage ratio for non-partitioning transposition sorts is 1. For quicksort, the storage ratio is $(N+\log N)/N$. For radix exchange, the storage ratio depends upon g, the range of the item keys, and is $(N + \log g)/N$ [16].

It has been shown [11] that the Shell sort is an $O(N^{3/2})$ process if the Hibbard modification is used, and that both radix exchange and quicksort are $O(N\log N)$. These results take into account both comparisons and item transfers. The other transposition sorts are all $O(N^2)$ as can be seen from Table 8 [16].

Table 8. Analysis of Transposition Sorts

	Comparisons		Transpositions	
	Max	Min	Max	Min
Normal	$(N-1)^2$	$N-1$	$N(N-1)/2$	0
Bubble	$N(N-1)/2$	$N-1$	$N(N-1)/2$	0
Alternating	$(N-1)^2$	$N-1$	$N(N-1)/2$	0
Funnel	$N(N-1)/2$	$N-1$	$N(N-1)/2$	0
Even-odd	$(N-1)^2$	$N-1$	$N(N-1)/2$	0
Successive Minima	$N(N-1)/2$	$N(N-1)/2$	N	0
Ranking	$N(N-1)/2$	$N-1$	$N(N-1)/2$	0

There are available [10] results as to the expected number of comparisons and transpositions, assuming unique data items, for some of these algorithms. For the bubble, funnel, and even-odd transpositions sorts, the expected number of comparisons is approximately:

$$((N^2-N-2)/2) + (N+1)\ z(N+1) - N\ z(N)$$

where $z(N)$ is approximately:

$$(\pi N/2)^{1/2}$$

The expected number of comparisons for the ranking sort is:

$$((N^2+7N)/4) - 1.6 - \log N - (1/2N)$$

For all the algorithms in Table 8 except the successive minima sort, the expected number of transpositions is:

$$N(N-1)/4.$$

This completes the analysis of sorting algorithms. The criteria of analysis have been explained and results presented. In deciding upon an algorithm for a particular application, characteristics of the application--including the type and amount of data, the amount of core memory

available and the characteristics and number of serial files available--must be considered as well as the relative efficiency of algorithms. The ability of sorting algorithms to make use of information regarding the input data has been discussed. In the following chapter, the problem of using available information about the data to achieve more efficient sorting will be examined.

Chapter 4

Some Improvements

Much of the published literature dealing with sorting techniques and their analysis is restricted to one particular frame of reference, that is, the problem of sorting lists of items which are non-redundant and randomly distributed in value. When the sorting problem is approached from this perspective, resultant algorithms are not limited in terms of the types of data which can be successfully sorted. Such generality is a desirable characteristic and algorithms which operate efficiently within this framework have been developed. If one limits the class of problems then more efficient algorithms can be developed.

One of the processes done in sorting is the gathering of information about the data. It would appear reasonable that if a priori information is known, it could be used to simplify the process of information gathering, and hence achieve more efficient sorting.

The purpose of this chapter is to consider the sorting problem with respect to data about which there is some known information. This will involve first a definition

of the different forms of information which may be available. Secondly, methods will be presented for using the different types of information for more efficient sorting.

4.1 Types of Information

Information about data to be sorted may be of two general forms. The first to be considered is 'a priori' information, or that which is available at the outset of the sort procedure. The second is information gained during the sorting process. Specifically of interest is that which is gained through comparisons.

4.1.1 A Priori Information

Generally, there may be a priori information about the distribution of the values found in the data list or about partial orderings present in this list. Consider first the case of a priori information regarding distribution. This may take the form of knowledge of the existence of redundant items. Two cases in this category which will be considered are a knowledge only of the number of unique items and the stronger case in which not only is the number of unique items known, but also, there is some information as to how the redundancies may be distributed. In addition, a priori information about the probability distribution of

the population from which the items in the list are drawn is considered.

Three types of a priori information regarding partial orderings will be considered. The first is the knowledge of the existence of sublists of ordered items. Second is knowledge of the existence of sublists such that, for every pair, all items in one sublist bear a given relation to all items in the other sublist. The third type of a priori information regarding partial ordering is the knowledge of a bound on the distance any item may be from its final position.

4.1.2 Information Gained Through Comparisons

Even though it may be that no a priori information regarding the distribution or positioning of data values is available, it is obvious that, for lists of size greater than three, every item need not be compared with every other item in order to gain enough information to place the items in sorted order. This is largely due to the fact that the information gained through, say, x comparisons can extend beyond simply a new found knowledge of the order relationships between x pairs of items. This is so because of the transitivity of the ordering relation. This characteristic is discussed in section 4.3.

4.2 Use of a Priori Information

4.2.1 Knowledge of Redundancies

In considering the problem of sorting lists containing redundancies, first to be discussed will be the lower bound on the number of comparisons required. Henceforth in this discussion it will be assumed that sequence preservation is not required. The theoretical lower bound of N log N comparisons required to sort N items was derived under the assumption that all N items are unique. If, however, there are $M < N$ unique items, this lower bound is too high.

In order to determine the lower bound for data lists with redundancies, the number of unique output strings that could result from the application of a sorting procedure must first be established. For N unique data items there are obviously N! unique permutations of the output string. However, if there are N items in total, of which only $M < N$ are unique, then some of these N! permutations are equivalent.

One possible, but incorrect, argument as to the number of unique output permutations that may result from sorting N items of which M are unique is as follows. Assume that with each unique item, i, there is associated a

'replication factor', ri, such that ri is the number of times the i-th item appears in the list. Observe that if the value x occurs 3 times, then for any given arrangement of the N-3 other items, there are 3! different ways of placing the three x values in the remaining three positions. However, all 3! ways will produce identical output orderings. From this observation it is easily argued that the number of unique output permutations that may result from application of a sorting procedure is $N!/(r1!r2!...rM!)$. This argument, while it does indicate the number of unique permutations of N items of which M are unique, is incorrect since they cannot all be possible results of a sorting procedure.

In order to understand why these cannot all be possible output permutations, one should note that any set of identical input values must always be permuted to adjacent final positions. There are N! possible permutations only if any pair of input items can be permuted to non-adjacent output locations and therefore, when redundancies are present, there are not N! possible output orderings.

At this point, note should be made of two relevant facts. The first is that, given any set of items in which there are redundancies, all equivalent items can be considered as one item for the purpose of calculating the number of unique and possible output permutations. There are

two reasons for this. These items will always be permuted to adjacent output locations. The permuations that vary only in the order in which these items are placed in a particular set of adjacent output locations all result in the same output string. Secondly, the order of appearance of items in the output string is dependent upon the relationships between the unique values. There are M! possible orders of appearance, or output permutations, of the M unique items.

Given a knowledge of only N and M, there are many ways in which the redundancies can be distributed. Let R be a vector of length M such that each component is ≥ 1 and their sum is N. When R is associated with a vector of M unique objects, it specifies a redundancy distribution.

Consider a vector X of M unique objects and some R, associated with X. Let Y be a permuted vector of X and associate R with Y also. Both the sets of objects defined by X and R and by Y and R have M! possible unique sorted permutations. But it is clear that they are isomorphic. Therefore only one of the sets need be considered when computing the number of unique and possible output permutations. This isomorphism exists whenever two redundancy vectors, which are equal as ordered partitions of N, are applied to the same set of unique objects. Note that if the redundancy vectors, R1 and R2, are unique ordered partitions of N, this isomorphism between sets of M!

permutations does not exist. Therefore, if $W = f(N,M)$ is the number of ordered partitions of N into M parts, there are $M! \times W$ unique and possible output permutations.

For example, consider the case in which $N=7$ and $M=4$. There are $W=3$ distinct (in an unordered sense) ways of distributing the redundancies. The possibilities are:

1) 1 item appears 4 times
   3 items appear 1 time
2) 1 item appears 3 times
   1 item appears 2 times
   2 items appear 1 time
3) 3 items appear 2 times
   1 item appears 1 time

For each of these there are $M!=4!$ possible and unique output permutations. So, in total, there are $3 \times 4!=72$, as opposed to $7!=5040$, possible and unique output orderings.

Given this information regarding the number of outputs that need be considered, the lower bound on the number of comparisons required can be established. It is reasonable to use here a tertiary rather than a binary computation tree due to the fact that equality of items is being considered as well as simply < and >. This implies that a comparison can yield a result of <, =, or >. For such a tree, with $M! \times W$ leaves, the depth of the tree must be at least $\log(M! \times W)$ and therefore at least that many tertiary

comparisons are required to be able to sort any N numbers of which M are unique. However, since $\log_3 x < \log x$ and for the sake of comparison, throughout the rest of this discussion a binary computation tree will be assumed.

It is interesting to consider the behaviour of W with respect to N and M. Note that W is equal to the number of ordered partitions of N-M into at most M parts. For N>>M, a reasonable estimate of W is [5]:

$$(N-M-1)! \;/\; ((M-1)!(N-2M)!M!).$$

Thus, using a binary computation tree, the theoretical lower bound on the number of comparisons required is:

$$\log (N-M-1)! - \log (M-1)! - \log (N-2M)!$$

Since M<<N, this lower bound is much less than log N!.

Results as to the value of W are also available [5] for the case in which $M \le N \le 2M$. For this case, the number of ordered partitions of N-M into at most M parts is equal to the number of ordered partitions of N-M. It has been shown that log W approaches

$$\pi(2(N-M)/3)^{1/2}$$

asymptotically. Exact values of W for $0 \le N-M \le 100$ are also available [5]. Consider the case in which N-M = 32. The value of W is 8,349 and the ratio of:

$$(\log (W x M!)) \;/\; (\log N!)$$

for different values of N and M, assuming a binary computation tree, are given on the next page.

M	N	Ratio
32	64	.44
50	82	.56
60	92	.60
68	100	.65

It has been established that the theoretical lower bound on the number cf comparisons required to sort N items with redundancies is less than that for sorting N non-redundant items. Because such a difference in the theoretical lower bound exists and because sorting is often carried out on lists which contain redundancies, the problem of deriving sorting algorithms which operate efficiently on such lists is now considered.

First consider the quicksort algorithm. Recall that this is a partitioning algorithm in that, at each iteration, lists are divided into 3 sublists. The iteration procedes until a list of length 2 occurs. These items are compared and placed in sorted order. If there is a sublist of length $k>2$ in which all items are the same value, and if the normal iteration procedures were carried out, there would be $k-2$ more subdivisions of this sublist, requiring $(k+1)(k-2)/2$ comparisons. However, none of these comparisons is necessary as the sublist of size k is in place and sorted. Therefore the iteration procedure should have been halted for this sublist. The problem, then, is to recognize at which point the normal iteration should be halted. This

is simply a problem of determining when a sublist contains only equivalent data items.

Generally it is not desirable to check every sublist, before dividing it into two sublists, to see if all items are equivalent. The a priori knowledge of redundancies may be used to determine an optimum sublist size at which to start checking for equivalent values. If, for instance, it is known that most of the duplicated items will appear between 3 and 5 times, it then is reasonable to check all sublists of size 5 to see if all items are equal. Note that if only 2 unique values are found in a sublist then, at the next iteration, all items will be in place if the minimum valued item is used as a median estimate (assuming 'less than or equal to' and 'greater than' comparisons).

If the checking of sublists is done for lists of size k or less then k must be such that the number of comparisons required to determine equality of all items in the sublist (i.e., k-1 comparisons) is significantly less than the number of comparisons required to complete the iteration process (i.e., (k+1)(k-2)/2). On the average, if the sublist contains items of different values, then k/2 comparisons have been wasted.

A second class of sorting algorithms that may be modified to operate more efficiently are the address

calculation sorts. The major difference lies in the fact that the address calculation function need only distribute items M-ways rather than N-ways. If a linked list structure is used, then information about the redundancies could be used to determine when items which were distributed into the same bucket should undergo another distribution phase and when they should be checked for equivalence. Information regarding the distribution of redundancies is also useful in deriving a bin technique suitable to the data to be sorted and thus requiring less adjustment.

In both these cases, the more information that is available about the possible distribution of redundancies, the easier it is to produce a more efficient algorithm. If it is known precisely how the redundancies will occur then an address calculation sort can be used to sort the list in one distribution phase. However, an extension of what has been discussed in this section is the case in which the probability distribution of the population from which the items in the list are drawn is known or can be well estimated.

4.2.2 Knowledge of Probability Distribution

A knowledge of the probability distribution of the data values can easily be used to achieve efficient sorting. For example, with such information, an address sort for

which the expected number of data moves is very small could be derived. This, however, implies a storage ratio of at least 2 and, if the redundancies are not distributed uniformly, many comparisons may be necessary. A method is now proposed for sorting large data lists containing much redundancy. This method has a storage ratio of approximately $(N + 2M)/N$, where M is the number of unique items.

Assume that the list to be sorted consists of items coded in a Huffman code, so that the items appearing most frequently have the shortest code words. Basically, a downward radix exchange sort is used to place all equivalent items into groups. After the i-th iteration, all code words of length i are in place and need not be considered further. Upon completion of the sort, groups of items could then be output in the desired order.

The important variables, subroutines and conventions used in the algorithm are as follows:

Variables

1. STRING
    - STRING is a linked list of variable length with the maximum number of entries equal to the number of code words.
    - Entries in STRING are of the form:
      INDEX,REP,POINTER
      where:
      INDEX--is an index into the list to be sorted.
      REP--is used for two purposes:
      a) to indicate when the items in the sublist pointed to by INDEX are in sorted position, and

b) when the items are in sorted position, REP is incremented at each iteration for the purpose of proper indexing into STRING.

- Initially STRING contains one entry, which is given as input:
  1,0,blank

2. START
   - START is a pointer to the first entry of STRING.
3. S
   - S is used to indicate the number of entries in STRING.
   - Initially S=1
4. STP
   - STP is used as a pointer into STRING.
   - Initially STP=0
5. I
   - I is the iteration counter.
   - Initially I=0
6. TABLE
   - TABLE contains as entries all code words arranged in ascending order with respect to length. Those code words of equal length are in ascending order on value.
   - The entries are of uniform length and a non-binary character indicates the end of each word.
7. T
   - T is a pointer into TABLE.
   - Initially T=1
8. LIST
   - LIST is the list of N items to be sorted.
   - Items are stored in entries of uniform length, which is determined by the length of the longest code word.

<u>Subroutines</u>

1. SPLIT
   - SPLIT has input parameters I, Q, R and output parameter L.
   - does a binary radix exchange on items from positions Q to R in LIST, on the I-th bit (high to low order).
   - $Q \leq L \leq R+1$ is the position in LIST of the start of the 1's sublist.
2. GETSPACE
   - GETSPACE creates a new record of the same format as an entry in STRING. The record returned is temporarily named NEWENT.

Conventions

1. A(x) refers to the address of x.
2. INDEX(J), REP(J), POINTER(J) refer to the INDEX, REP, or POINTER field, respectively, of the J-th entry in STRING.
3. The symbol '=' is used for assignment.
4. .GE.,.NE.,.EQ.,.LE., and .GT. are logical tests for 'greater than or equal', 'not equal', 'equal', 'less than or equal', and 'greater than'.
5. In order to eliminate needless detail, STRING is indexed as a table, rather than as a linked list.

Algorithm

Given the variables as described, the proposed algorithm is as follows:

```
1. Initialize
   - I=1
   - S=1
   - STP=0
   - T=1
   - START=A(INDEX(1))
2. If STP.GE.S
   THEN go to 13
   ELSE STP=STP+1
3. If REP(STP).NE.0
   THEN REP(STP)=2xREP(STP)
        go to 2
4. If INDEX(STP).NE.blank
   THEN J=0
        go to 7
5. GETSPACE
   NEWENT(1)=INDEX(STP)
   NEWENT(2)=REP(STP)
   NEWENT(3)=POINTER(STP)
   POINTER(STP)=A(NEWENT)
6. S=S+1
   STP=STP+1
   go to 2
7. If POINTER(STP+J).EQ.blank
   THEN R=N
        go to 9
   ELSE J=J+1
8. If INDEX(STP+J).EQ.blank
   THEN go to 7
   ELSE R=INDEX(STP+J)-1
```

```
 9. Q=INDEX(STP)
    SPLIT(I,Q,R,L)
    GETSPACE
10. If L.NE.Q
    THEN go to 12
    ELSE NEWENT(1)=blank
         NEWENT(2)=0
         NEWENT(3)=A(INDEX(STP))
11. If STP.EQ.1
    THEN START=A(NEWENT)
         go to 6
    ELSE POINTER(STP-1)=A(NEWENT)
         go to 6
12. If L.EQ.R+1
    THEN NEWENT(1)=blank
         NEWENT(2)=0
         NEWENT(3)=POINTER(STP)
         POINTER(STP)=A(NEWENT)
         go to 6
    ELSE NEWENT(1)=L
         NEWENT(2)=0
         NEWENT(3)=POINTER(STP)
         POINTER(STP)=A(NEWENT)
         go to 6
13. J=0
14. If TABLE(T) of length I
    THEN J=J+1
         PNTR(J)=TABLE(T)+1
         T=T+1
         go to 14
15. If J.EQ.0
    THEN go to 19
    ELSE SUM=0
         K=1
         P=1
16. If REP(K).EQ.0
    THEN SUM=SUM+1
    ELSE SUM=SUM+REP(K)
17. If PNTR(P).EQ.SUM
    THEN REP(K)=1
         P=P+1
    ELSE K=K+1
         go to 16
18. If P.LE.J
         THEN K=K+1
         go to 16
19. J=1
```

```
20. If REP(J).NE.0
    THEN J=J+1
    ELSE I=I+1
         STP=0
         go to 2
21. If J.GT.S
    THEN STOP
    ELSE go to 20
```

The major loop in this algorithm is from statements 2 to 21. Each time this loop is started, I is incremented. This loop contains four loops of importance. The loop from statements 2 to 12 check the following for all entries, J=1,2,...,S, in STRING:

1) if, in statement 3, REP(J).NE.0

   then the items in the sublist pointed to by INDEX(J) are in sorted position and need not be split.

2) if, in statement 4, INDEX(J).EQ.blank

   then some code word or words are missing and the entry is maintained only for indexing purposes.

3) if neither 1) nor 2) is true

   then the sublist pointed to by INDEX(J) is split and a new entry is added to STRING so that there is an entry for the 0 and for the 1's sublist.

The loop in statement 14 checks TABLE for code words of length I and, for each, stores its value+1 in PNTR. The loop from 16 to 18 finds entries in STRING pointing to the code words of length I and sets REP for these entries to 1, indicating that the sublist is sorted. The last loop, from

20 to 21, checks if REP is non-zero for all entries in STRING. If so, the list is sorted.

A brief description of the Huffman coding scheme [9] will further clarify the algorithm. The Huffman coding system results in a set of variable length code words such that, for any two items to be coded, say x,y, where $p(x)<p(y)$ (where $p(x)$ is the probability of x), the length of the code word for y is less than or equal to the length of the code word for x. In other words, the more probable the item, the shorter is its code word. In the case to be considered here, the code words are binary strings. Another important characteristic of the code words derived is that they are comma free. In other words, if 10 is a code word, no other code word starts with 10. Thus, after I iterations, all equal code words of length I are grouped together and no code word of greater length can be in the group. A characteristic of minor importance is that, if the length of the longest code word is, say, 7 then every binary number $b<2^6$, is either a code word or the first part of a code word. For such a code to be derived it is necessary, of course, that the probabilities of the values to be coded be known.

Before discussing the advantages of a sorting procedure as has just been described, it should be noted that this procedure is not suggested for general use in any

sorting problem. An underlying assumption is that the probability distribution of the population from which the items are drawn is known or can be closely approximated. Note also that this procedure is suggested for use in cases for which the number of unique items is relatively stable and much less than N.

One advantage of such a procedure is generality. That is, no matter what the population distribution, the algorithm is applicable and need not be altered for different distributions. This is due to the fact that the code itself contains much information regarding the distribution of values.

Perhaps the most important advantage of this procedure is the fact that the most probable values are placed in sorted position earliest. This is because the most probable values have the shortest codes. Once placed in sorted position, these items are not dealt with in subsequent iterations. Therefore, only a small number of items, the least probable, are handled in all iterations.

Also to be noted is the fact that complete sets of items with identical values are placed in sorted position at the same iteration, for any size of set. Not only are they placed in sorted position but it is easily recognized when they are in position and need not be considered further.

This is different than many of the commonly used sorting algorithms for which items are positioned one at a time and for which it is difficult to detect if, in fact, a group of items are all of equivalent value and all in sorted position.

Finally, this procedure has the advantage of a constant bound on the number of iterations. If there are N items to be sorted and M code words, as the size of N grows, with M constant, the upper bound on the number of iterations required remains constant.

Another possible advantage of this procedure is the fact that single bit comparisons are used. At each iteration, binary splitting of sublists is done on the basis of the value of a single bit, rather than on the value of an entire item. The characteristics of the computer used determine whether single bit operations are faster.

One question to be considered is whether or not existing algorithms can be as efficient as the proposed algorithm, given the same amount of information about the distribution of values. In many cases, the algorithms cannot be altered to take advantage of this information, as was discussed in Chapter 3. One algorithm which could make use of such information is quicksort.

Given a priori information regarding the probablity distribution of the population from which the items in the list were chosen, near perfect median estimates could be derived for use in the quicksort algorithm. The algorithm would then require approximately log N iterations to sort N items. At each iteration, comparisons of entire items, rather than of single bits, are made. As N grows, the number of iterations required will grow.

Consider the problem of sorting 64 items, of which 13 are unique with probabilities of occurrence as shown in Table 9 [9].

Table 9. Example of Huffman Code

Item	Probability	Code	# of Occurrences
A	.20	10	13
B	.18	000	11
C	.10	110	6
D	.10	111	6
E	.10	011	6
F	.06	0101	4
G	.06	00100	4
H	.04	00101	4
I	.04	01000	3
J	.04	01001	2
K	.04	00110	2
L	.03	001110	2
M	.01	001111	1

Quicksort would require approximately log 64 = 6 iterations. The number ofcomparisons required at each iteration are approximately as shown on the following page.

Iteration	Comparisons
1	64
2	63
3	61
4	57
5	49
6	33
	327

Because the size of the longest code word is 6, the proposed algorithm would require 6 iterations as well. However, the number of comparisons required for each iteration with the proposed algorithm are as follows:

Iteration	Comparisons
1	64
2	64
3	51
4	22
5	18
6	3
	222

Therefore, the total number of comparisons required for the proposed algorithm is 222 as opposed to 327 for quicksort. Note also that these are bit-wise comparisons rather than full item comparisons. If the number of unique items were to remain constant while the number of items in the list became 128, the proposed algorithm would still require 6 iterations and quicksort would require 7. Approximately 444 comparisons would be required for the proposed algorithm, while quicksort would require 786.

4.2.3 Knowledge of Ordered Sublists

Another possible form of a priori information that may be available regarding the data is the knowledge of the existence of sorted sublists within the list. If it is known that such sublists do exist, then certainly it is more efficient to make use of the information by choosing an algorithm that will not 'waste time' with ordering groups of items that are already in sorted order.

Given a knowledge that the list contains sorted sublists, certain questions must be considered. For instance, are the sublists of approximately equal length or is there a great variation in the size of the sublists? If the sublists vary greatly in length, is it only because there are some very small sublists while the rest are of approximately equal length? Another question to be considered is whether or not the delimiting points of the list into sublists are known. Finally to be considered is whether or not sorting is to be done in core.

Most random-access sorting algorithms cannot be easily adapted to make use of the fact that the list to be sorted contains ordered sublists. The natural merge sort, however, is well adapted to this type of data. Merge sorting is most efficient when all lists at the same level of the merge are of approximately the same length [15]. Therefore,

if it is known that there is a great variation in the lengths of the ordered sublists, then the shorter lists should be merged until all lists are of approximately the same length. Note that the sequence preserving characteristic is lost. If delimiting points of the sublists are not known, N-1 comparisons will yield this information. If all the sublists, except those which are very small, are of approximately the same length, then the small sublists may either be placed into a non-trivial sublist or may themselves be sorted to form a larger sublist. Which of these two approaches is the more efficient depends upon the number of single items and the size of the average non-trivial sublist. Assume there are r single items and the average size of a non-trivial sublist is s. To place the items into sorted position in non-trivial sublists would require approximately r log s comparisons. To order the single items would require approximately r log r comparisons.

If a serial merge sort is used for such data, and if it is expected that there will be great variation in the sizes of the strings, then a replacement sort procedure is recommended. This will serve to produce longer strings [15] and should tend to minimize the number of very short strings.

4.2.4 Groups Ordered with Respect to Each Other

It may be known that the data list consists of r groups of contiguous items, L(1),L(2),...,L(r) such that, for any i,j, all items in group L(i) bear a certain relation (e.g., ≤, ≥) to all items in group L(j). In such a case, it would appear inefficient to sort the list of N items paying no attention to the known relationships between items that exist. If, for instance, the list consists of r groups of size p then, for every item, its relationship with (r-1)p other items is known. A reasonable approach, then, would be to sort separately the groups of unordered items in proper order. To sort these groups would require approximately $r \times p \log p = N \log p = N (\log N - \log r) = N \log N - N \log r$ comparisons. To sort the list as a whole would require approximately N log N comparisons. Since the groups were sorted one at a time, the extra storage space required would be the same as that required for a list of size p, rather than for a list of size N. Therefore a saving in storage is also achieved in this way.

This saving in the number of comparisons required and in storage space required is less when the number of groups is small. Also, if the size of the groups is variable then, the closer that the size of the largest group is to N, the smaller is the saving in time and space. However, the savings in comparisons and storage space can be extremely

significant in certain cases and the programming effort required to effect a more efficient algorithm are minimal.

4.2.5 A Bound on Distance

In this section, consideration will be paid to the case in which the known information about the data to be sorted is in the form of a bound on the distance an item can be from its sorted position. An algorithm that can be easily adapted to make use of this type of information is the Shell sort. This algorithm is efficient largely because, with the initial choice of a large increment, an item can be moved a large distance towards its final position with only one comparison.

If there is a bound on the distance an item can be from its final position then this bound may be used to decrease the size of the initial increment as was explained previously. The choice of a smaller initial increment results in the saving of one or more pass. It has been shown [11] that, for the Shell sort, the saving of one pass is as desirable as saving (10/9) N data moves.

If the known bound on the distance that an item can be from its final position is r and if the natural sequence of increments is d(1),d(2),...,1 then this series can be modified to start at d(i) such that d(i+1) is less than r

and d(i) is greater than or equal to r. Such a modification results in a saving of both comparisons, and perhaps, data moves. By simply using one less pass, up to N/2 comparisons can be saved. Any data moves that may have resulted from the passes which are deleted from the sort procedure are unnnecessary since it is known that an item is not a distance greater than r from its final position. These unnecessary data moves are also thereby avoided.

The first pass made with such a choice of initial increment need not follow the normal bubbling procedure. For instance, if two items a,b are compared and interchanged then neither a nor b need be involved in another comparison in the initial pass as both have been moved a distance greater than or equal to the bound on which an item may be from its final position. The subsequent passes will be as for the normal Shell sort procedure. Therefore this algorithm is easily modified to make use of such information and to effect savings in both comparisons and, in some cases, data moves.

4.3 Use of Information Gained Through Comparisons

One aspect of the problem of using information to gain more efficient sorting that has been discussed [1] is that of making the most of information gained through comparisons. Given N data items, there are N!/(N-2)!2!

possible binary comparisons [1]. However, in the course of sorting it will probably be decided that some of these comparisons are not necessary. A comparison is unnecessary if it is possible to calculate in advance the result of the comparison, i.e., the algorithm must make use of the transitivity of the order relation to determine the result of the comparison [1].

In studying the phenomenon of applying the transitivity relation in order to decrease the number of comparisons required, there have been [1] some definitions specified. Given that an algorithm can be defined as a series of comparisons of the form (a,b) where a and b are two items being compared, a <u>prediction</u> is defined as a series of 3 comparisons of the form (a,b);(b,c);(a,c). If it is found that $a<b$, $b<c$ or that $a>b$, $b>c$ then the result of the comparison (a,c) is known and the comparison need not be made. Therefore, a prediction is defined as <u>active</u> over a set of data if either of these cases holds. It should be noted that, while previous analysis [1] assumes that these comparisons are consecutive, it is possible to consider a set of 3 comparisons as a prediction, whether or not this comparison series is interspersed with other comparisons. If a(i) denotes the i-th of n data items, then a general form

of a prediction can be defined. This generalized form is the series of comparisons:

$$(a(1),a(2));(a(2),a(3)) \ldots (a(k-1),a(k));(a(1),a(k))$$

and the comparison $(a(1),a(k))$ is actively predicted if either:

$$a(1)<a(2)<\ldots<a(k-1)<a(k) \quad \text{or}$$

$$a(1)>a(2)>\ldots>a(k-1)>a(k).$$

Note that equality can be included in the definition of active prediction.

A set of predictive algorithms has been defined [1] and comparison made of their relative efficiency (the average number of comparisons made). This analysis is not relevant here as most commonly used sorting algorithms are not included in the set of predictive algorithms. The important point is that the ability of an algorithm to recognize predictions and to alter the comparison series when they are active, can result in greater efficiency.

Two cases of specific interest indicated by the analysis previously described will now be considered. The first concerns sorting lists containing redundancies. If, during the series of comparisons, it is found that A=B then any comparisons later called for between B and an item with which A has been compared or vice versa need not be carried out, as the results of such comparisons have been actively predicted. Thus, an algorithm making such comparisons is

less efficient that an algorithm in which such unnecessary comparisons are not made. While this is an obvious conclusion, many algorithms do not utilize information concerning equality of items. Non-partitioning transposition sorts and the ranking sorts are algorithms which do not incorporate prediction of this type.

The second case to be considered is an example of an algorithm in which best use was not made of the information gained in previous comparisons. The Shell sort, as originally defined, used a series of increments, $d(1), d(2), \ldots, 1$, where $d(1) = N/2$ and $d(i+1) = d(i)/2$. In the first iteration $a(1)$ was compared with $a(N/2+1)$. Assume it was found that $a(1) < a(N/2+1)$. In the second iteration $a(1)$ is compared with $a(N/4+1)$. Assume that $a(1) > a(N/4+1)$. The next comparison called for was $a(1)$ with $a(N/2+1)$ which is needless. Similar cases can be found throughout the original Shell sort algorithm. It was for this reason that the Shell sort was modified with respect to the choice of increments and, thus modified, the algorithm proved to be more efficient.

The concept of predictions has been introduced and it has been pointed out that efficiency with respect to the utilization of information is important in achieving efficient sorting. While inefficiencies may be 'hidden' in the algorithm, it is worthwhile to examine algorithms for

the purposes of determining if inefficiency in the comparison procedure is present. This may be due to neglecting to make use of the transitivity of the ordering relation or it may simply, as with the Shell sort, be a problem of making comparisons that have already been made. In any case, inefficiency is due to failure to recognize when a comparison need not be made.

In this chapter the problem of making use of information to achieve more efficient sorting has been discussed. This information may either be a priori or may simply be that which is gained through comparisons. In either case, it has been shown that more efficient sorting can be achieved by making full use of the information available.

Chapter 5

Conclusion

The primary classification of sorting algorithms is on the basis of whether or not serial-access files can be used throughout the procedure. Those algorithms intended for implementation using serial-access files are referred to as serial sorts and can be further classified as merge, column or partial pass algorithms. Those algorithms intended for implementation using random-access files are referred to as random-access or internal sorts and can be further classified as distributive, merge, tree structure, or strictly comparative techniques.

The efficiency of serial sorts is evaluated in terms of the number of passes over the input list required for a fixed number of files. It has been shown that single phase techniques are more efficient than double phase techniques, and that partial pass techniques are often the most efficient. While merge sorts can take advantage of sorted order in the input list, no serial algorithms can be readily adapted to utilize information regarding such order or the distribution of key values.

Random-access sorts are evaluated with respect to the number of comparisons and data moves required and to the storage ratio. Tree structure and merge techniques cannot be easily modified to make use of information regarding ordering or duplications. Neither of these classes include algorithms whose efficiency increases significantly when duplications are present. The class of strictly comparative algorithms includes algorithms that take advantage of the sorted order of the input list. Some strictly comparative algorithms can be modified to utilize information regarding distribution of values. Distributive algorithms do not take advantage of sorted order. While radix sorts cannot utilize information regarding the distribution of key values, address calculation sorts are easily modified to achieve more efficient sorting when such information is available.

Information regarding the data to be sorted may be available at the outset or may be gained, throughout the sorting procedure, by means of comparisons. A priori information regarding the distribution of data values or regarding ordering present within the list may be available. It has been shown that the number of comparisons required to sort a list of items with redundancies is less than for a list of non-redundant values. The problem of utilizing information about the data to gain more efficient sorting has been discussed and some solutions presented. Since, for many sorting applications there is some known information

about the data, the problem of utilizing information to achieve more efficient sorting is worthy of future study. The most important problem is that of developing techniques for efficiently sorting data lists containing redundancies. This involves the problem of altering the comparison series to avoid repetitious comparisons when two items are found to be equal. Such 'set oriented' algorithms would be of great value for many sorting applications. The development of algorithms specifically for defined forms of input data is also of value, in that, with such tailored algorithms, gains in sorting efficiency can be made.

References

1. Beus, H, L., 'The Use of Information in Sorting', <u>J.A.C.M.</u> , Vol. 17, No. 3 (July, 1970), 482-495.
2. Brooks, F. P. Jr., and Iverson, K. E., <u>Automatic Data Processing System/360 Edition</u> , (John Wiley and Sons, Inc., 1969).
3. Flores, I., <u>Computer Sorting</u> , (Prentice-Hall, Inc., 1969).
4. Gale, D., and Karp, R. M., 'A Phenomenon in the Theory of Sorting', <u>I.E.E.E. Conference Record of the 1970 11 th Annual Symposium on Switching and Automata Theory</u> , 51-59.
5. Hall, M. Jr., <u>Combinatorial Theory</u> , (Blaisdell Publishing Co., 1967).
6. Hibbard, T. N., 'An Empirical Study of Minimal Storage Sorting', <u>C.A.C.M.</u> , Vol. 6, No. 5 (May, 1963), 206-213.
7. Hildebrandt, P., and Isbitz, H., 'Radix Exchange - An Internal Sorting Method for Digital Computers', <u>J.A.C.M.</u> , Vol. 6, No. 2 (April, 1959), 156-163.
8. Hoare, C.A.R., 'Quicksort', <u>Computer Journal</u> , Vol. 5, No. 1 (1962), 10-15.

9. Huffman, D. A., 'A Method of Construction of Minimum Redundancy Codes', Proc. IRE , Vol. 40, No. 10 (September, 1952), 1098-1101.

10. Iverson, K. E., A Programming Language , (John Wiley and Sons, Inc., 1962).

11. Knuth, D. E., The Art of Computer Programming , Vol. 3,(Addison Wesley Publishing Co., to be published).

12. Lawler, E. L., 'The Complexity of Combinatorial Computations: A Survey', Proceedings of 1971 Polytechnic Institute of Brooklyn Symposium on Computers and Automata , (Polytechnic Press, 1971), 305-312.

13. Lorin, H., 'A Guided Bibliography to Sorting', IBM System Journal , Vol. 10, No. 3 (1971), 244-254.

14. Malcolm, W. D. Jr., 'String Distribution for the Polyphase Sort', C.A.C.M. , Vol. 6, No. 5 (May, 1963), 217-220.

15. Martin, W. A., 'Sorting', Computing Surveys , Vol. 3, No. 4 (December, 1971), 147-174.

16. Rich, R. P., Internal Sorting Methods Illustrated with PL/1 Programs , (Prentice-Hall, Inc., 1972).

17. Rivest, R. L., and Knuth, D. E., 'Bibliography 26. Computer Sorting', Computing Reviews , Vol. 13, No. 6 (June, 1972), 283-289.

18. Shell, D. L., 'A High Speed Sorting Procedure', C.A.C.M. , Vol. 2, No. 7 (July, 1959), 30-32.

19. Sobel, S., 'Oscillating Sort - A New Sort Merging Technique', J.A.C.M. , Vol. 9, No. 3 (July, 1962), 372-374.